AD-A107 645 NAVAL UNDERWATER SYSTEMS CENTER NEWPORT RI F/G 9/2
INTERCOMPUTER TRANSPORTATION OF ASSEMBLY LANGUAGE SOFTWARE THRO--ETC(U)
OCT 81 D L BRINKLEY
UNCLASSIFIED NUSC-TR-6004 NL

1 OF 1
AD A
107645

END
DATE
FILMED
1 -82
DTIC

2

NUSC Technical Report 6004
15 October 1981

Intercomputer Transportation of Assembly Language Software Through Decompilation

Donald L. Brinkley
Combat Control Systems Department

Naval Underwater Systems Center
Newport, Rhode Island / New London, Connecticut

3

PREFACE

This work was accomplished under NUSC Project No. J47503, "Combat Control System Technology Improvements," principal investigator - Thomas Conrad (Code 3511) and NAVSEA Task No. 18335. The sponsoring activity is the Naval Sea Systems Command, program manager - V. Whitaker (PMS-409).

This report was initially submitted to the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology in May 1980 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science.

The technical reviewer for this report was G. M. Hill (Code 351).

REVIEWED AND APPROVED: 15 October 1981

[illegible]

J. C. Lamb
Head, Combat Control
Systems Department

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 6004	2. GOVT ACCESSION NO. [illegible]	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) INTERCOMPUTER TRANSPORTATION OF ASSEMBLY LANGUAGE SOFTWARE THROUGH DECOMPILATION		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Donald L. Brinkley		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Underwater Systems Center Newport Laboratory Newport, Rhode Island 02840		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NUSC Project No. J47503
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Sea Systems Command Washington, DC 20362		12. REPORT DATE 15 October 1981 13. NUMBER OF PAGES 41
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION / DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Assembly Language Software	Intercomputer Translation
Decompilation	Automatic Translators
Software Portability	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Portability of assembly language software can be achieved in a practical way only through the use of automated translators. The fact that translation between two representations at the same semantic level is a clumsy and difficult task has frustrated the development of such technology. This clumsiness stems from the fundamental differences in the architectural structures of different machines. In its simplest form, the translation process actually becomes the process of simulating one machine or another. The goal of such translation, however, should be the generation of assembly language code for the target machine that approximates, in efficiency and in appearance, code written to perform the same task by a "good" assembly language programmer in the target language.

A translator that performs a decompilation of the source program into an intermediate representation at a higher semantic level is described. Code in the target assembly language can later be generated from this intermediate representation. This translation scheme is shown to remove most of the machine dependency from assembly language software..

TABLE OF CONTENTS

A

LIST OF ILLUSTRATIONS

INTERCOMPUTER TRANSPORTATION OF ASSEMBLY LANGUAGE SOFTWARE THROUGH DECOMPILATION

INTRODUCTION

MOTIVATION AND DIRECTION

Software portability is a primary concern of users anticipating an upgrade or replacement of computer hardware currently in use. If the software to be preserved is written in some standard, high-level language that both systems support through compilers, it can be shared in a straightforward manner. However, if the high-level language in which that software is written is not supported by both systems, or if the software is written in the assembly language of one of the machines, transporting the software between computer systems becomes a problem. Automatic intercomputer translation of assembly (or machine) language code is a possible solution to this problem.

The most obvious application domain for a translator is that of debugged, fully-implemented programs. Such a translator has greater potential utility than this, however. Consider the translation of an assembly language program that is not yet fully implemented but has no errors in syntax. Let this translation be realized textually. (In other words, let both the input and output programs of the translator be humanly readable code written in the assembly languages of their respective machines). If the output program is as well commented and mnemonically meaningful as the input program, then it is possible to continue the debugging process with the output program. The translation can be realized at any point in the development process as long as the input program is syntactically correct. This capability would be particularly effective in a situation where, for some reason, the target machine supported debugging better than the source machine. (For example, the former might support some online debugging tool.) Also, looking at the semantics of the program from a new perspective sometimes reveals features (and misfeatures) not otherwise noticeable. For instance, extraneous data motion can often be detected after a change in perspective. (It is assumed here that program documentation is preserved in the transition from source to target code.)

The utility of such a translator should not be denied even by one who abhors the idea that assembly language programs are still being written. There seems to be, however, a paucity of written matter and research concerning this subject. Probably the foremost reasons for the lack of real technological progress in this area are the varying architectural structures of different computers and the difficulties involved in producing efficient code for one machine from code intended for a quite different machine.

The UYK-7 computer is used here as an example of a source machine for which the use of this type of translator is particularly appropriate. The UYK-7 is a militarized version of a 32-bit, ones complement Sperry Univac machine. Produced in the late 1960s - early 1970s, it is used widely for U.S. Naval applications. However, its relatively aged architecture makes it a likely candidate for replacement in the near future. Since a great deal of assembly language software has been generated for this machine, a translator of the sort described herein could aid substantially in reclaiming much of the investment in UYK-7 assembly language software.

This report considers the possibility of Digital Equipment Corporation's PDP-11/70 computer or a similar machine as a successor to the UYK-7. It is worthwhile to examine some of the more pronounced architectural differences between the UYK-7 and the PDP-11/70 for two reasons: (1) an intercomputer translator for the two machines would have to address these differences, and (2) the utility of such a translator would be constrained by these differences.

TRANSLATION CONSTRAINTS

Instruction Formats

The difference in the machines that is most relevant to the designer of a translator is the difference in instruction formats. The PDP-11/70 assembly language has a standardized format consisting of an operator and one or two operands, with data descriptors associated uniformly with each operand. The UYK-7 has several formats that can be used. For instance, in one UYK-7 instruction format, the operator, accumulator register, bit field within the 32-bit word, index register, and memory base register must be specified. These formats accompany the operators rather than the operands because of the accumulator, single-operand structure of the UYK-7. Translation between instruction formats is a minimal requirement of such a translator.

Word Size

The UYK-7 computer has a word size of 32 bits, while the PDP-11/70, a minicomputer, has a word size of 16 bits. The word size affects the size of integer numbers each machine can manipulate, as well as the precision of real numbers. While the limits of the integers that the UYK-7 can handle are sufficient for most integer calculations in practice, the PDP-11/70 user is limited to signed, single-precision integers no larger than 32,768. Since the UYK-7 programmer has made the assumption that 32 bits can be used to represent each integer, a well-designed translator would have to provide for that.

A UYK-7 to PDP-11/70 translator could use double precision (two words) for all its integer values. As an alternative, the translator could request human intervention to specify, for each integer variable, whether to use single or double precision and generate code accordingly. When the user does not specify either single or double precision, however, double precision could be assumed.

There is no practical way to maintain the precision of real numbers, since both machines already use double precision representation for them. The two words with which the UYK-7 represents real numbers provide 64 bits of precision, while two words on the PDP-11/70 only provide 32 bits of precision.

Processing Power

The UYK-7 processor hardware is generally more powerful than that of the PDP-11/70. For example, the UYK-7 has a single machine level instruction that computes the 32-bit square root and the 32-bit residue of a 64-bit integer; the PDP-11/70 does not offer an equivalent instruction. However, the hardware of the UYK-7 can usually be simulated in software on the PDP-11/70. Although the execution time for the functionally equivalent sequence of instructions in this case is greater on the PDP-11/70, it is possible to execute that sequence, and the translator should account for it.

Console Features

As a part of its instruction set, the UYK-7 provides for the testing of certain switches on the actual console of the machine. For instance, there is an instruction JC (Jump Conditional Setting) in the UYK-7 repertoire. This instruction jumps to a given position in the program, conditional on the (up or down) position of a switch on the console. The PDP-11/70 hardware is not equipped to handle such an instruction or a semantic equivalent (without halting the process and waiting for information to be entered from switches on the console, if any exist). Therefore, translation cannot include such instructions.

Data Access

Another difference in the two machines involves the structure of the memory. Although memory protection is not handled precisely the same way in each, it is handled analogously. Protected memory can be accessed by a privileged program in both, and the semantics of this sort of access can be captured by the translator. Having privileges in the UYK-7 to execute an instruction that references protected memory makes it necessary to have analogous privileges in the PDP-11/70 for the translation to succeed.

The PDP-11/70 has six general purpose registers that can be used for operations on data. Many of these operations can act on data in regular memory locations as well, but execution time is greatly reduced by action on the data in registers. On the other hand, the UYK-7 has eight accumulator registers, at least one of which must be used for a typical operation. Since the reasons for moving data into registers are different in the two machines, more efficient code would be produced if the translator moved data into the registers by using criteria other than what the source program dictates.

The virtual memory for a program on the UYK-7 is partitioned into eight segments, any one of which may be referenced by a program through the use of segmentation registers. Although the PDP-11/70 supports no such segmentation, this does not present a problem to the translator. A program's virtual memory on the PDP-11/70 can be artificially divided the same way through software created by the translator. In the case of a textual translation, the translator can generate unique symbols that correspond to the eight segments and suffix the name of a variable with those symbols identifying the segment that contains it.

Indexing is accomplished in the the UYK-7 by the use of a set of seven index registers. In the PDP-11/70, it is accomplished by using the general registers. Translation of an indexed variable is therefcre straightforward; however, the six general registers are put in greater demand. Intelligent register allocation becomes even more important.

One more point is relevant in a discussion of the differences in data accessing on the two machines. Even though a PDP-11/70 word is two bytes in length, memory is addressed in bytes; that is, each byte has its own address. Even numbered addresses begin words. Although this addressing scheme presents no real problems to the designer of a translator, it should be kept in mind during the design.

Integer Representation

The UYK-7 is a ones complement machine, while the PDP-11/70 is a twos complement machine. This fact is relevant to the translator only when the instruction LN (Load Negative) is used. When this instruction is used by a programmer, it is not clear whether the negative of some integer or the bitwise complement of that data word are to be loaded since they are expressed the same way on the UYK-7 but are distinct on the PDP-11/70. A consistent choice in translation is the only reasonable solution to this problem. Of course, each such application of this assumption should be noted by the translator to facilitate human identification of instances where the assumption is incorrect.

Condition Codes

The UYK-7 has condition codes that allow the following facts to be recorded after execution of a COMPARE instruction: "not equal versus equal," "less than versus greater than or equal," and "within limits versus outside limits." The PDP-11/70 condition codes support the first two pairs of relations, but not the limits relation. However, this relation can be expressed in software by using the first two relations and comparing the data with the limits in several steps.

Another difference in the two computers' uses of condition codes relates to the circumstances upon which the codes are changed. The UYK-7 only changes the condition codes after execution of a COMPARE instruction. However, many PDP-11/70 instructions change them (e.g., ADD and SUB). No translation problem occurs when the PDP-11/70 is used as the target machine if the codes are examined by the translator immediately after simulating the UYK-7 COMPARE instruction.

Traps, Interrupts, and Input/Output

Traps, interrupts, and input/output (I/O) are handled very differently in the two machines. For example, the UYK-7 uses a special set of commands for I/O control that initiate output buffers, set specific bits in the processor, and accomplish other machine dependent functions. The utility of automatic translation breaks down at this point. One way of treating this problem that should be considered, particularly in the case of textual translation, is simply to clearly mark on the output listing, each instruction that is not or cannot be translated. If such instructions are indicated by the translator, the user can then patch them with code generated by hand. This plan is practical only if program comments are preserved and the code that is generated by the translator is understandable.

Memory Size

Because of the difference in word sizes in the two machines, considerably less memory is addressable on the PDP-11/70 than on the UYK-7. This limitation, which is fundamental, is caused by the target machine, not the translator; the translator cannot, therefore, accept the responsibility for it.

TRANSLATION SCHEMES

There are two general schemes for accomplishing a translation of the type described previously. One of the schemes requires the use of a mapping of the source language instruction set directly into that of the target language. The other involves decompilation, or inverse compilation, of the assembly language code.

The former scheme functions on the premise that each instruction in the source language can be equivalently represented by an instruction or sequence of instructions in the target language. In order to effect a translation by this scheme, the instruction map must first be realized for the general case. If the implementation of this scheme is limited to the mapping, the target machine essentially simulates the hardware architecture of the source machine. Before translation, the program is tailored (in a sense, "optimized") for execution on the source machine. Effecting an instruction by instruction translation of this program directly to the target language seems rather impractical in light of the significant architectural differences of different machines. The basic structure of a program written in the assembly language of one machine generally differs from that of a program written in the assembly language of another machine. For instance, one machine may use a single accumulator for all arithmetic operations. In that case, sequences of instructions frequently take the general form of:

```
Load X
Operate Y
Store Z.
```

Creating code of that form through the translator for a machine with six general purpose registers, and which can operate directly on data in memory, introduces great inefficiency. The goal of an intercomputer assembly language translator is to generate a program that looks very much as if it were written by a "good" assembly language programmer for the target machine. Therefore, an optimizing process for this translated code would enhance the translator's utility.

When incorporating optimization, care must be taken that: (1) each instruction in the source program translates into an exact semantic equivalent in the target program, and (2) through whatever optimizations are brought about, the semantics of the program are left unchanged. If it can be shown that the semantics of each instruction in the source program are preserved through the mapping into sequences of instructions in the target program and also through optimization, translation correctness can be deduced.

As an aid in evaluating this translation scheme, a partial mapping of the instruction set of the AN/UYK-7 computer directly into that of the PDP-11/70 was accomplished. In all, 110 UYK-7 instructions were mapped into PDP-11/70 equivalents. Even though this mapping was somewhat unpolished, rough ideas of efficiency can be obtained by analyzing it.

In generating the mapping, there were often tradeoffs between clarity, space (memory) requirements, and (execution) time requirements of the target code. When these arose, they were resolved according to the following priorities: (1) clarity, (2) space requirements, and (3) time requirements.

The mapping was achieved only by imposing some restrictions on particular instructions. For example, translation of the UYK-7 instruction, which computes the square root of a 64-bit integer, generates PDP-11/70 code, which only computes the square root of a 32-bit integer. Such compromises are necessary for the reasons cited above.

The 110 instructions that were mapped were chosen because they appeared to be simple to map. Most of the instructions in the UYK-7 repertoire that were not mapped are machine-dependent instructions relating to traps, interrupts, and I/O.

The mapping did not include any PDP-11/70 instructions added for optimization purposes, such as movement of data into the registers simply for more optimal execution time. Additionally, no UYK-7 instructions were mapped to PDP-11/70 subroutine calls exclusively in order to achieve greater space efficiency.

Results of the instruction mapping are as follows:

1. Out of 110 UYK-7 instructions mapped into PDP-11/70 instructions, there were 93 that mapped into less than eleven PDP-11/70 instructions each. The mean number of PDP-11/70 instructions generated by a UYK-7 instruction in this group was 4.3 with a standard deviation of 2.4.

2. Out of the remaining 17 UYK-7 instructions that mapped into greater than ten PDP-11/70 instructions, the mean was 15.1 PDP-11/70 instructions generated per UYK-7 instruction with a standard deviation of 5.4. The largest number of PDP-11/70 instructions generated for any UYK-7 instruction was 30 for the integer square root instruction.

3. Overall, with 110 instructions mapped, the mean number of PDP-11/70 instructions generated per UYK-7 instruction was 6.0 with a standard deviation of 5.0.

These results were somewhat predictable from the computer differences described previously. The translation efficiency can be improved by using the decompilation scheme, which will be described, or by adding an optimization step to the translator.

The alternate translation scheme involves decompilation of assembly language, which is by nature at a rather low semantic level, into a functionally equivalent intermediate representation at a higher level. Decompilation extracts the semantics of a program by translating a machine-oriented language into a procedure-oriented language. After decompilation, this scheme concludes with generation of the target code from the intermediate procedure-oriented representation. (See figure 1.) The use of a higher level intermediate representation provides both machine independence and an opportunity for manipulating groups of instructions within a greater context. In the case of textual translation, it is possible for programmers to continue the development effort at this level. The higher level of representation afforded through decompilation promotes the generation of efficient target code.

The decompilation system described can be viewed as an instruction mapper to an intermediate language and an associated optimizer. This scheme is in sharp contrast to a scheme in which instructions of one computer are mapped directly to instructions of another without the use of an intermediate representation. Use of an intermediate representation in computer translation offers another advantage, particularly if several computers must share code. If M computers need to share code with N more computers, (M*N)-(M+N) translators are saved by the use of an intermediate representation. (See figures 2 and 3.) This phenomenon has been discussed before, and an attempt was made to create a universal language into which all other languages could be compiled and from which machine-dependent code could be generated. That project was called UNCOL.[1] It failed to achieve its goals because it could not represent completely the semantics of diverse high-level languages or the variety of machine architectures. The choice of an intermediate representation is very important, and is discussed more fully in the section dealing with Pass Two.

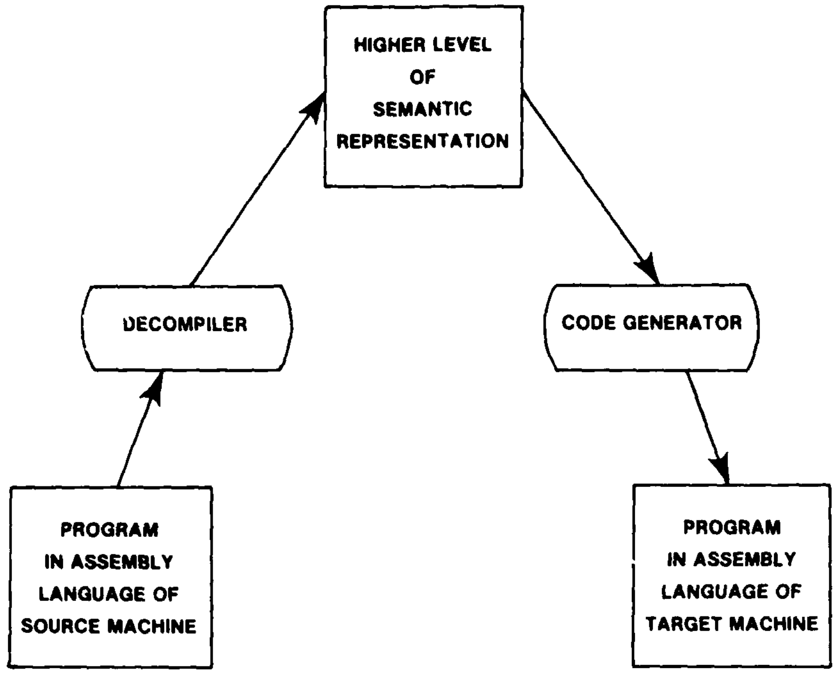
HIGHER LEVEL OF SEMANTIC REPRESENTATION
DECOMPILER
CODE GENERATOR
PROGRAM IN ASSEMBLY LANGUAGE OF SOURCE MACHINE
PROGRAM IN ASSEMBLY LANGUAGE OF TARGET MACHINE

Figure 1. Program Translation Using Decompilation

THE PROTOTYPE

The decompiling phase of an intercomputer assembly language translator has been implemented. The source language on which it acts is a subset of ULTRA/32, the assembly language of the AN/UYK-7 computer. The system has been given the name ZEBRA. The intermediate representation language, called STRIPE, is comprised of quadruples:

Operator	Operand 1	Operand 2	Destination

and associated comments. Also produced by the decompiler is a symbol table and a control flow description of the decompiled program. The output of ZEBRA is textual (that is, humanly readable), thus allowing program development to continue. (See figure 4.) PASCAL is the language of implementation for ZEBRA.

The decompilation effort, as presently implemented, requires three passes through the various forms of the subject program. Pass One scans the source program, lexically analyzing and parsing it. Two output files are produced: a listing file indicating detected errors, and a file containing the program in a fixed format (PASCAL records), which is more convenient for analysis by later passes. Also generated by Pass One are tables describing the flow of control of the program and containing information about the symbols used in the program.

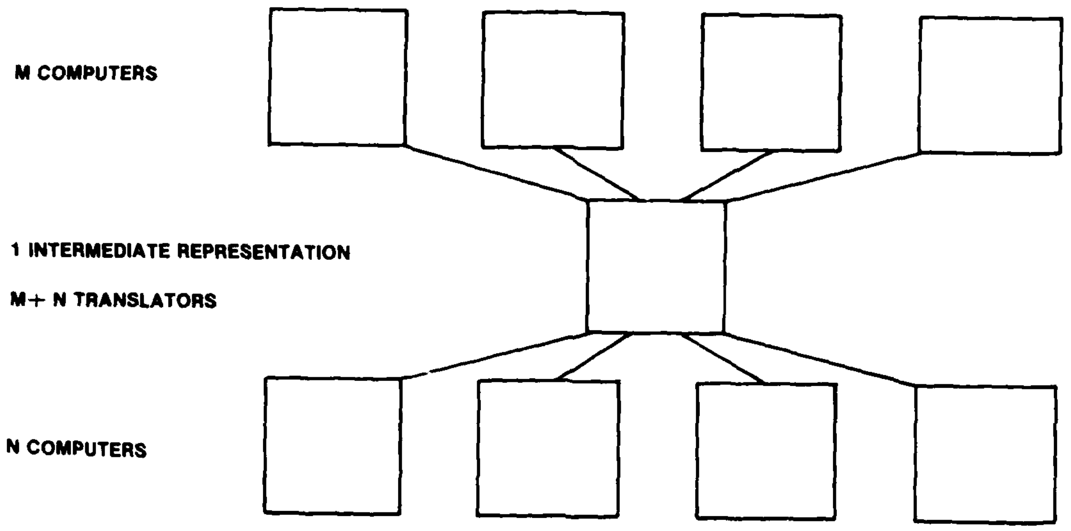

Figure 2. Computer Translation with an Intermediate Representation

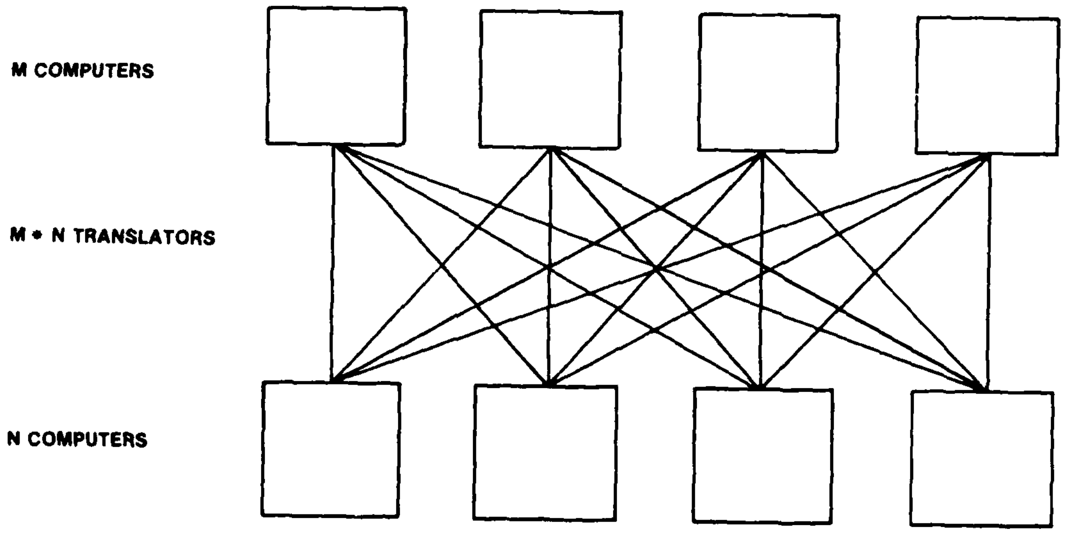

Figure 3. Computer Translation without an Intermediate Representation

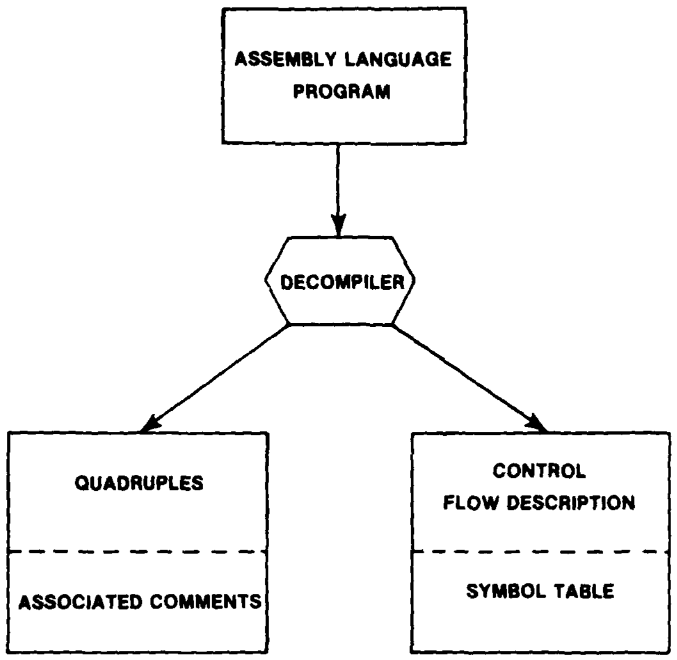

Figure 4. The Zebra Compiler

Pass Two translates the program from PASCAL records to the intermediate language STRIPE. A few errors are detectable at this stage that could not be conveniently detected earlier.

Finally, Pass Three compresses the STRIPE representation, where possible, by eliminating extraneous LOAD and STORE instructions. This step accomplishes the decompilation itself. Decompilation is limited to this extent in order to maximally capture the semantics of assembly (low-level) code. Housel,[2,3] in reporting on a decompiler that translates from Knuth's MIX Assembly Language into PL/1, described this need to choose carefully the level of decompilation. 'The current state of the art has not completely eliminated the "special case" problem. However, the extent of automatic translation can generally be increased if the level of the target translation is sacrificed. The target language must be capable of expressing lower level semantic constructs (e.g. shift, mask, etc.). The capability is usually found in systems programming languages but is not included in the more common algebraic languages (e.g. FORTRAN).' Since translating to an existing high-level language is not the goal of this report, but rather translating to another low-level language, the level of decompilation chosen here seems both necessary and sufficient.

RELATED WORK

The problems of machine portability in general have been investigated for many years. Some of the following approaches have been suggested for solving these problems: high-level languages, interpretation, emulation, hand-recoding, compiler-compilers, and machine-independent code generators. Warren[4] discussed several of the previous topics relative to software portability.

Other attempts at attaining machine independence at various levels of automatic computer language translation have been made. Richards[5] described a portable compiler for the language BCPL, as well as OCODE, the language used to specify the interface between the machine-independent and machine-dependent parts of the compiler. He was somewhat successful in transporting the language BCPL (a block-structured language resembling ALGOL 60) between computers. He described his experience as follows:

> BCPL has now been transferred to between ten and twenty different machines using *OCODE as the* interface between two halves of the compiler. The time taken to complete a transfer is very variable depending strongly on the computing facilities available, and usually takes between three and five months if the implementer has no previous knowledge of BCPL and no access to the donor machine, but it may take as little as three or four weeks in ideal conditions. Much depends on how long it takes to design and implement the interface with the operating system; this can be very little if one chooses to provide the user with only a few very primitive facilities, but it is usual to give him a more powerful interface and this necessarily requires much more work.

Machine independent code generation was analyzed by Newcomer[6] and Miller.[7] Each of these researchers produced notations for formally describing the hardware architecture of a computer and a language. UNCOL,[1] as mentioned previously, was a much heralded but unsuccessful attempt to create a universal language. The Mobile Programming System[8,9] also relied on translation to an intermediate macro language. It differed from the UNCOL approach in that the macro language was tailored specifically to the primary source language with which the designers worked.

Relatively little research has been conducted in the area of decompilation. However, several projects are noteworthy. A few words should be said about decompilation as it exists in the literature of computer technology. Every decompiler system studied in this research translates a subject program from a machine (or assembly) language into an existing high-level compiler language. Program portability is the goal in these systems. The problem with such decompiler systems is that they attempt too high a level of decompilation to be practical. One advantage of ZEBRA lies in its simplicity.

Barbe's PILER system[10] is an example of a decompiler that has a target level of translation higher than that of ZEBRA. The scope of the software captured by PILER is limited by the semantic level of its translation target. It is a powerful and flexible decompiler, but requires substantial user interfacing. The translation source language is a machine language formally described to the system as an input. The target language of the translation is specified within the PILER system as defined at translation time but remains at a rather high level. PILER relies on intensive bookkeeping to derive the program flow during analysis of source code. ZEBRA is similar in spirit with respect to its analysis but differs in its choice of translation level.

Hollander[11,12] proposed a strategy for decompilation which uses a table-driven, syntax directed metasystem for process description. The metalanguage he developed is quite general, and his techniques should be applicable to a large variety of operations. Housel and Halstead's work[2,3] has already been mentioned. The methodology they developed was important in the design of ZEBRA. However, their target level of decompilation was much higher than is necessary for simple intercomputer translation of assembly language software.

It is apparent that compiler technology lends itself quite well to decompilation. The syntax-directed translation suggested by Hollander is an example of this fact. Aho and Ullman[13] and Gries[14] provide a overview of compiler technology. Their methodologies for determining busy conditions (called "ud-chaining" by Aho and Ullman and defined in the Pass Three section) are utilized in Pass Three of ZEBRA. Additionally, the schemes they described for conducting a lexical analysis were helpful in designing Pass One.

PASS ONE - LEXICAL AND FLOW ANALYSIS

This section describes methodologies used in implementing Pass One of a decompiler from ULTRA/32, the UYK-7 assembly language, to STRIPE, an intermediate language designed for this purpose. Pass One scans the source program, lexically analyzing and parsing it. Two output files are produced: a listing file indicating detected errors, and a file containing the program in a fixed format (PASCAL records) more convenient for analysis by later passes. Also generated by Pass One are tables describing the flow of control of the program and containing information about the symbols used in the program.

LEXICAL ANALYSIS - SEPARATING DATA FROM INSTRUCTIONS

One of the functions of Pass One is to perform a lexical analysis of the source program. This analysis identifies the program's instructions and data. A mapping of the program is generated which indicates whether a particular line contains an instruction or data. This instruction mapping, called the INSMAP, describes to later decompilation phases how the contents of the output file of the previous phase are to be interpreted. The instruction mapping is implemented in PASCAL as

```
PACKED ARRAY [1..MAXPROGSIZ] OF BOOLEAN
```

where MAXPROGSIZ is a program-defined constant indicating the maximum number of lines accepted for a source program. The instruction mapping is indexed by program line numbers. A TRUE value for a particular element of the mapping indicates that line contains either a valid instruction or no instruction (i.e., just a comment). A FALSE value indicates that data is defined and perhaps initialized on that line.

The output file of Pass One contains all the information that the source file contains. However, in the output file, all implied subfields of instruction operands are made explicit, and the components of the line are explicitly placed into PASCAL record format.

The Pass One output file is defined in PASCAL as a FILE OF PARSREC.

The complete definition of PARSREC follows:

```
    CONST
        SLINSIZ = 80;              (* Maximum number of characters in
                                     each source line *)
        STBLSIZ = 100);            (* Size of the symbol table *)
    TYPE
        SYMNUM = 1..STBLSIZ;       (* Symbol numbers indexing into the
                                     symbol table *)
        LCHARTYP = 1..SLINSIZ;     (* Index of characters across the
                                     source line *)
        REGNUM = 0..7;             (* Register numbers *)
        ADRTYP = (INSTR,DATA);     (* An address contains either
                                     instructions or data *)
        LINTYP = PACKED ARRAY
                 [LCHARTYP] of CHAR; (* One source line *)
        OPERATOR = (HLT, LA, AA,   (* These operators(using the
                    ANA, SA,C,       ULTRA mnemonics) are
                    JNE, JE, JG,     supported at this time -
                    JGE, JLT,        easily expandable to the
                    JLE, J, NOOP,    full instruction set *)
                    NONE);
        OPANDTYP =                 (* These are the different operand
          RECORD                     subfields *)
            A,K,B,S, : REGNUM;     (* Eight of each of these registers *)
            M,P : 0..31;           (* Additional fields used in some
                                     instructions - number of bits to
                                     shift, etc. *)
            Y : SYMNUM             (* A symbol table entry is made for
                                     each unique occurrence of Y *)
          END;                     (* OPANDTYP *)
        PARSREC =                  (* The format of output for Pass One *)
          RECORD
            COMMENT : LINTYP;      (* Each line can have an
                                     associated comment *)
            CASE ADRTYP OF;        (* The instruction map contains
                                     ADRTYP information *)
             DATA : (VAL : INTEGER); (* Initialization value *)
             INSTR:
               (OPCODE : OPERATOR;
                OPERANDS : OPANDTYP)
             END;                  (* PARSREC *)
```

The advantage of converting each line of source code into PARSREC is one of convenience for Pass Two. It is true that the translation to

STRIPE which Pass Two performs could have been done concurrently with Pass One, but the resulting code would have been much less modular and harder to debug and extend.

The ULTRA/32 operators presently supported by ZEBRA, which are discussed in this report, are now listed with their meanings:

HLT	Halt the process.
LA	Load an accumulator with the contents of a memory location.
AA	Add the contents of a memory location to an accumulator.
ANA	Add the negative of the contents of a memory location to an accumulator.
SA	Store the contents of an accumulator into a memory location.
C	Compare the contents of an accumulator with the contents of a memory location and set the condition codes accordingly.
JNE	Jump to a specified location if the condition codes reflect a "not equal" condition.
JE	Jump to a specified location if the condition codes reflect an "equal" condition.
JG	Jump to a specified location if the condition codes reflect a "greater than" condition.
JGE	Jump to a specified location if the condition codes reflect a "greater than" or "equal" condition.
JLT	Jump to a specified location if the condition codes reflect a "less than" condition.
JLE	Jump to a specified location if the condition codes reflect a "less than" or "equal" condition.
J	Unconditional jump to a specified location.
NOOP	No Operation.

In general, an ULTRA/32 instruction has associated with it more than one operand subfield. For instance, the LA (Load Accumulator) instruction has the following operand subfields:

A : 0...7;	Specifies which of eight accumulators is used.
Y : Symbolic;	Specifies the data memory location.
K : 0...7;	Specifies which fields of the 32-bit Y operand is used. (i.e., address, first half word, second half word, whole word, or one of the four bytes)
B : 0...7;	Specifies which Index Register is used. (A B-field of 0 indicates no indexing.)
S : 0..7;	Specifies which Base Register is used. (Segmentation is the memory scheme.)

If any of these subfields is omitted from the source coding line, the default value is 0. The Y subfield of every instruction is assumed to be symbolic by ZEBRA. In other words, absolute addresses (numerical) are not handled by ZEBRA. Absolute addresses on one computer, in general, have no meaning on another. The ULTRA/32 assembler recognizes the Y subfield as a number of words of displacement from the contents of the Base Register indicated in the S subfield. So, two uses of the same symbolic Y subfield name with different S subfields would be viewed by the assembler as two different variables. One (presently unimplemented) way of accounting for this fact in the decompiler is to append the S subfield value symbolically to the name in the Y subfield before entering the name into the symbol table. This action, if consistently taken, resolves any naming ambiguities.

The Y subfield is represented in the output file of Pass One as a symbol number, that is, an index into the symbol table. A symbol's name is placed into the symbol table and, thus, given a number as soon as the lexical analyzer discovers it in the label field of some line or in the operand field of some instruction. However, no other information concerning the symbol is retained until it is seen in the label field of a line and the thing it labels identified.

As an example of an instruction in a Pass One input file and its corresponding entry in the output file, consider the following instruction:

```
LINELABL      LA        1,LOC,,5    . This is a comment.
```

The PARSREC entry corresponding to this instruction contains the following component values:

```
OPCODE    < = LA;
OPERANDS
                .A <=  1;
                .Y <=  2;
                .K <=  0;
                .B <=  5;
                .S <=  0;
                .M <=  0;
                .P <=  0;
COMMENT <= "This is a comment."
```

Two entries are made into the symbol table: one for "LINELABL" and one for "LOC". The symbol "LINELABL" is given symbol number 1, and "LOC" is given symbol number 2. Note that the Y component of PARSREC takes on as its value the symbol number of "LOC".

The symbol table is defined in PASCAL as an

ARRAY [SYMNUM] OF STBLENT

The complete definitions of SYMNUM and STBLENT follow:

```
    CONST
      STBLSIZ = 100;            (* Size of the symbol table - may be
                                  arbitrarily changed *)
      MAXSYMSIZ = 8;            (* Maximum number of characters
                                  allowed in a symbol name *)
      MAXPROGSIZ = 600;         (* Maximum total number of lines in a
                                  program *)
    TYPE
      LINENO = 1..MAXPROGSIZ;   (* Line number of program from
                                  beginning *)
      SYMNUM = 1..STBLSIZ;      (* Symbol numbers indexing into
                                  the symbol table *)
      SYMARR = PACKED ARRAY
              [1..MAXSYMSIZ]
               OF CHAR;         (* String which holds the symbol name *)
      SYMTYP = (IMMED,SIMPVAR,  (* Type of Symbols - constants, simple
               ARRVAR,SIMPTRAN,   variables, arrays, simple transfers
               NOTYP);            (instruction labels) *)
      STBLENT =                 (* Each symbol table entry contains
                                  this information*)
        RECORD
           NAME : SYMARR;       (* Name of the symbol *)
           CASE TYP : SYMTYP OF (* If the type of the
                                  symbol is *)
             IMMED :            (* Immediate *)
              (VAL : INTEGER);  (* Then it has a value *)
             SIMPVAR, ARRVAR,
             SIMPTRAN :         (* For these types, it has
                                  an address *)
               (ADDR, SIZE :
                LINENO)         (* Size only used for array variables *)
        END;                    (* STBLENT *)
```

A symbol can be one of four different types: simple variable, array variable, constant, or simple transfer (instruction label). In order to be a valid symbol, it must appear once in the label field of some line. When that line contains a valid operator in the operator field, the symbol must represent the destination of a JUMP instruction and thus is of type SIMPTRAN. The phrase EQU in the operator field of a line indicates that the label is a constant (type IMMED). Finally, an integer in the operator field indicates that the symbol is a variable and the integer is the initial value. The type is ARRVAR, if more initial values follow on succeeding unlabeled lines; it is SIMPVAR, otherwise. NOTYP is used as an initial type for symbols whose names have been discovered in the operand field of an instruction but have not yet been seen to label a line. An error condition is indicated if the type of any symbol is NOTYP at the end of Pass One. In fact, the symbol type information is used primarily for detection of certain errors.

The symbol table contents after scanning the instruction

```
    LINELABL  LA 1,LOC,,5 .  This is a comment.
```

are described here:

```
    SYMBOL NUMBER <= 1
       NAME <= "LINELABL";
       TYP  <= SIMPTRAN;
       ADDR <= Current line number;
    SYMBOL NUMBER <= 2
       NAME <= "LOC";
       TYP  <= NOTYP;
```

The TYP value of NOTYP for symbol number 2 indicates that symbol has not yet been found to label a line. Also, that symbol has no valid address in the symbol table for the same reason.

CONTROL FLOW ANALYSIS

In order to decompile to a higher semantic level, it is necessary not only to identify a program's instructions, but also to determine how the instructions are related with respect to their execution sequences (i.e., the control structure). The methodology for determining the program flow of control is borrowed from compiler technology. Decompilation and compilation utilize control flow analysis for similar purposes. In both cases, flow analysis is used in determining the appropriateness of local optimizations. Since decompilation requires something of a local optimization, the effectiveness of flow analysis in decompilation is somewhat predictable.

The first step in flow analysis is to break the source program into basic blocks. In compilation, a basic block is a sequence of consecutive instructions that may be entered only at the beginning of the block. Once the basic block is entered, its instructions are executed in sequence without halt or possibility of exit from the block except at the end of the block. This definition is amended slightly for decompilation by specifying that the end of a basic block may contain more than one JUMP instruction as long as all but the last are conditional jumps and are conditional on the value of the same operand or pair of operands. This modification to the definition of basic blocks generally reduces the number of blocks contained in a program without sacrificing any information.

It is useful to portray basic blocks and their successor relationships by a directed graph called a flow graph. The nodes of the flow graph are the basic blocks, and the paths are specified by the presence or absence of JUMP instructions. The operands of all the JUMP instructions contained within a block define the immediate successors of that block.

The block (call it "B") beginning with the instruction physically following the last instruction of block A is the possible exception to this definition of immediate successors. Block B is counted among the immediate successors of block A if block A contains no unconditional transfer of control instruction and its conditional transfer of control instructions do not account for every logical case of the operands being tested. For example, there would be an implied transfer of control to the physically succeeding block of block A if the JUMP instructions of block A only included the "less than" and "equal" relations of one pair of operands. If the case occurs that "greater than" is true, then control is transferred to the physically succeeding block. The flow graph must reflect this information. Note that this method for determining the control structure is even effective on unstructured code.

Consider the following ULTRA/32 program:

```
(1)  LABL1   LA    1,XYZ
(2)          C     1,SOMEMEM
(3)          JG    LABL1
(4)          JE    LABL3
(5)  LABL2   J     LABL1
(6)  LABL3   LA    2,OTHERMEM
(7)          J     LABL2
```

The flow graph for this sample program is located in figure 5.

Block Table Description

Block bounds and the flow graph are organized into a block table in ZEBRA. Once again, the most convenient language in which to describe the structure of the block table is PASCAL. The block table is defined to be:

ARRAY [BLKNUM]OF BTBLENT

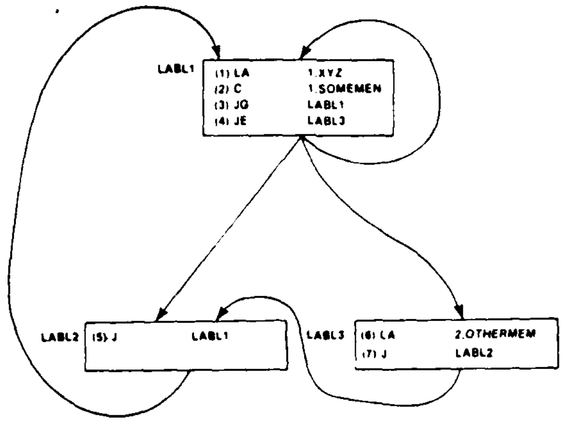

Figure 5. Sample Flow Graph

A description of BLKNUM and BTBLENT follows:

```
    CONST
        BTBLSIZ = 250;             (* Size of the block table - can
                                     be arbitrarily changed *)
        MAXPROGSIZ = 600;          (* Maximum number of lines of the
                                     source program - may be changed
                                     arbitrarily *)

    TYPE
        LINENO = 1..MAXPROGSIZ;    (* Number of lines since
                                     start of source program *)
        BLKNUM = 1..BTBLSIZ;       (* Block numbers - index
                                     into block table *)
        BINFLNK = ^BLKINFO;        (* BINFLNK and BLKINFO define
                                     a linked list of block
                                     numbers *)
    BLKINFO =                      (* Dynamic list of block numbers *)
        RECORD
          Next : BINFLNK;          (* A pointer to the next
                                     element of the list *)
                                     attributes associated with it *)
          DATA : BLKNUM;           (* The block number itself *)
        END;                       (* BLKINFO *)
    BTBLENT =                      (* Each block has the following
                                     attributes associated with it *)
        RECORD
          FINST, LINST : LINENO;   (* The line numbers of the first
                                     and last instructions contained
                                     in a particular block *)
          ISBAS : BINFLNK;         (* The base pointer to the list of
                                     immediate successor block
                                     numbers *)
        END;                       (* BTBLENT *)
```

The contents of the block table generated for the program whose flow graph is illustrated in figure 5 follows:

```
BLOCK NUMBER <= 1
    FINST <= 1;
    LINST <= 4;
    ISLST <= 1,3,2;
BLOCK NUMBER <= 2
    FINST <= 5;
    LINST <= 5;
    ISLST <= 1,
BLOCK NUMBER <= 3
    FINST <= 6;
    LINST <= 7;
    ISLST <= 2;
```

Note that the first and last instructions of a block are recorded in the block table as line numbers in the program. The immediate successor lists are lists of block numbers.

Block Table Generation Algorithm

A variety of circumstances combine to indicate the bounds of a basic block. The first instruction in a program begins the first block. Pass One linearly scans the program until the first instruction is found, initializes an entry in the block table, and assigns the current line value to be the first instruction of the new block. In every other case, the first instruction of a block is the first instruction following the last instruction of the physically preceding block. Note that in assembly language programs data often intervenes between basic blocks. So, the first instruction of a block may not necessarily fall on the line immediately following the last instruction of the physically preceding block but rather on the first line following where an instruction appears.

One circumstance that indicates the end of a block is the detection during scanning of an unconditional JUMP instruction, a HALT instruction, or conditional JUMP instructions which account for every logical case of the operands being compared. The current instruction being scanned when this circumstance is detected is the last instruction of the current block. The end of a block is also indicated by the detection of a non-JUMP instruction following conditional JUMP instructions that do not account for every logical case of the operands being compared. Another indication of the end of a block is the discovery that the label of the instruction line currently being scanned is the destination of a previously scanned JUMP instruction. In these two cases, the previous instruction is the last instruction of the current block, and the current instruction is the first of a new block. Other indications of the end of a block are discussed next.

The method for determining whether or not a sequence of conditional JUMP instructions accounts for every logical case of the operands being compared turns out to be quite simple. A condition value is preserved during Pass One of ZEBRA which reflects the status of the current location with respect to conditional JUMPs. Initially, before any JUMP instructions are detected, the value of the condition value variable is 0. As JUMP instructions with the following conditions are detected, the indicated increments are added to the condition value:

Condition	Condition Value Increment
<	1
=	2
>	4
> =	6
< >	5
< =	3
Unconditional	7

When the condition value reaches or exceeds 7, every logical condition of the variables being compared has been accounted for in that block's sequence of conditional JUMP instructions. For example, the sequence JL (Jump Less Than), JE, JG yields sequential condition values of 1, 1+2=3, 3+4=7. Hence, a block ends with the JUMP instruction that last incremented the condition value, and the condition value is reinitialized. A new block is also indicated if a non-JUMP instruction is scanned when the condition value is non-zero. This occurrence shows that some JUMP instructions were scanned, indicating that the current block should be ended, but that not all logical relations of the compared operands were accounted for. Therefore, an implied JUMP to this new block exists, and this block, beginning with this instruction, is an immediate successor to the block just ended. Note that the conditional JUMP instructions of one block are ensured to depend on the relationship of exactly two operands using this method. For more than the original two operands' relations to be involved in the conditional JUMPs, a COMPARE instruction would have to intervene (changing the condition codes). Since a COMPARE instruction is a non-JUMP instruction, this code would be interpreted by ZEBRA as indicating a new basic block.

The previous paragraph discusses one case for determining the immediate successor of a basic block. However, immediate successors are usually determined by their status as destinations of JUMP instructions. When a JUMP instruction is encountered in ZEBRA, the symbol table is searched to see whether or not the line labeled with the destination of the jump has been scanned. (It has not been scanned if there is no entry in the table for this symbol or if the type of that symbol is NOTYP.) If the line has indeed been scanned, then the block table is searched to determine to which block the destination belongs. If the destination is the first instruction of a block, that block becomes an immediate successor of the current block. Otherwise, the jump destination falls in the middle of some block, and that block must be split. Consider such a case in which an existing block is to be split into new blocks 1 and 2. First, the destination of the JUMP instruction becomes the first instruction of block 2 and the last instruction of the old block becomes the last instruction of block 2 (unless the destination of the jump is in the current block, in which case block 2 becomes the current block). Next, the instruction immediately preceding the destination becomes the last instruction of block 1 and the first instruction of the old block becomes the first instruction of block 1. Finally, block 2 becomes the immediate successor of block 1, and the immediate successors of the old block become the immediate successors of block 2.

If the line labeled with the destination of the JUMP instruction has not been scanned, then that symbol's name is first entered into the symbol table if it has not already been entered. Then, its symbol number is appended to a linked list data structure called the Unscanned Block List (if not already there). This structure is a list of the labels of the first instructions of all unscanned blocks which have been discovered thus far. Associated with each label in this list is a reference list of block numbers that contain a JUMP instruction whose destination is the line with this label. When a label is entered into the Unscanned Block List or the attempt is made for a label already contained in the list, the block number containing the JUMP instruction that references this unscanned block is entered into the reference list for that label. Hence, the Immediate Successor List of each block numbered in the reference list should include the block that contains the jump destination. The information kept in the Unscanned Block List is used and removed from the list when the line label of any instruction matches an entry in the Unscanned Block List. When an instruction is scanned for which that is true, it is known that a new block should start at that point. The current block is first ended with the previous instruction. Then, information gathering about a new block is begun in the block table; the new block gets a number. Finally, this new block number is added to the Immediate Successor Lists of each of the block numbered in the reference list of that entry in the Unscanned Block List.

INTERESTING EXCEPTION CONDITIONS DETECTED BY PASS ONE

Several conditions result in an exception being raised during Pass One processing. The UYK-7 is a ones complement machine, and therefore represents -0 differently from 0. Since most modern computers use twos complement representation, ZEBRA is justified in detecting and flagging occurrences of this constant in the source program. I/O and interrupt instructions, along with other unsupported instructions, are also detected and flagged by ZEBRA because of their machine-dependent nature.

JUMP instructions whose destination is indexed by some non-zero value indeterminate at translation time are not acceptable to Pass One. Since the index register can often take on unpredictable values, the destination of the JUMP instruction is often unpredictable. This case compromises the integrity of the entire flow graph produced by Pass One and thus cannot be allowed.

Indirection is also flagged, untranslated, by ZEBRA at this time. Since the point of indirection is indeterminate at the time of translation, its use as the destination of a JUMP instruction could unpredictably violate the flow graph in the same way as an indexed jump. However, another problem is introduced by allowing unrestrained use of indirection in general. Normally self-modifying code is detected by Pass Two by examining the operands of instructions which store data into memory. Self-modifying code is indicated if the destination of the STORE is an instruction location. Of course, the semantics of self-modifying code are highly machine dependent. Use of indirection inhibits Pass Two's ability to detect self-modifying code and, hence, is rejected by ZEBRA.

PASS TWO - TRANSLATION TO STRIPE

The task of Pass Two is to translate the formatted output of Pass One into STRIPE. Pass Two implements a one line to one line translation of the input file of strongly formatted ULTRA/32 to STRIPE (formatted in PASCAL records). The Pass Two translation of ULTRA/32 to STRIPE is carried out on this one-to-one basis as a precursor to the program reduction or optimization phase of ZEBRA (Pass Three). During this translation phase, the original source program comments and symbol names are preserved. This feature of the translation is important in providing for program modification and extension at the target level.

The basic structure of STRIPE was extracted from the structure of an intermediate representation often used for compilation, simple quadruples. During the design phase of ZEBRA, it was suggested that an existing formal intermediate level language such an UNCOL or P-code (an often used intermediate language in PASCAL compilation) might make a good choice for an intermediate language in ZEBRA, particularly since code generators already exist for these languages. However, use of those languages was rejected on the grounds of their inflexibility and inability to adapt to many source machine-target machine pairs. An important quality of the intermediate language in a decompiler is its ability to represent, succinctly, the semantics of a program independent of any particular machine.

The output file of Pass Two is defined to be a FILE of STRIPE. STRIPE has the following PASCAL definition:

```
CONST
    STBLSIZ = 100;           (* Size of the symbol table -
                               arbitrarily extensible *)
    SLINSIZ = 80;            (* Maximum number of characters
                               in a source line *)
TYPE
    ZSYMNUM = 0..STBLSIZ:    (* Symbol numbers as used in
                               the symbol table and
                               0 - no symbol *)
    LCHARTYP = 1..SLINSIZ;   (* Index of characters across
                               the source line *)
    STROPTOR = (HALT, ASN, ADD,
                SUB, JNE, JE,
                JG, JGE, JL,
                JLE, JMP, NONE);  (* STRIPE Operators - NONE
                               signals that no output
                               instruction is generated *)
```

```
ADRTYP = (INSTR, DATA);  (* An address contains either
                            instructions or data *)
LINTYP = PACKED ARRAY
        [LCHARTYP]of CHAR;    (* One source line *)
STRIPE =                 (* Format of output for Pass Two *)
  RECORD
    COMMENT : LINTYP;     (* Each line can have an associated
                            comment *)
    CASE KIND : ADRTYP OF;     (* Kind of line *)
    DATA : (VAL : INTEGER):    (* Data initial values are
                            preserved *)
    INSTR:
       (OPERATOR ; STROPTOR;  (* STRIPE operators *)
        FRSTOPAN, SECOPAN,
        DEST : ZSYMNUM)  (* Three operands *)
  END;                   (* STRIPE *)
```

Instructions in STRIPE take the form of

```
OPERATOR       OPERAND1       OPERAND2       DESTINATION
```

STRIPE operators are easily understandable. Following is a list of the STRIPE operators which are presently implemented along with a description of their meanings.

HALT	Halt the process.
ASN	Assign the contents of OPERAND1 to the DESTINATION.
ADD	Add the contents of OPERAND1 to the contents of OPERAND2 and place the result in the DESTINATION.
SUB	Subtract the contents of OPERAND2 from the contents of OPERAND1 and place the result in the DESTINATION.
JNE	Jump to the DESTINATION if the contents of OPERAND1 and OPERAND2 are "not equal."
JE	Jump to the DESTINATION if the contents of OPERAND1 and OPERAND2 are "equal."
JG	Jump to the DESTINATION if the contents of OPERAND1 are "greater than" those of OPERAND2.
JGE	Jump to the DESTINATION if the contents of OPERAND1 are "greater than" or "equal" those of OPERAND2.
JL	Jump to the DESTINATION if the contents of OPERAND1 are "less than" those of OPERAND2.
JLE	Jump to the DESTINATION if the contents of OPERAND1 are "less than" or "equal" those of OPERAND2.
JMP	Unconditionally JUMP to the DESTINATION.
NONE	No instruction.

Consider the ULTRA/32 instruction sequence which represents the PASCAL equation C := A+B:

```
        LA  1, A, 3         . 3 indicates the whole word.
        AA  1, B, 3
        SA  1, C, 3
```

That same sequence in STRIPE would be represented as

```
        ASN   A           <NULL>     $REGA1$$
        ADD   B            $REGA1$$  $REGA1$$
        ASN   $REGA1$$    <NULL>     C
```

Note that Accumulator Register 1 is represented in STRIPE by the temporary variable name $REGA1$$. This representation is explained in greater detail below. This form of representation is particularly well suited to simplification (as in Pass Three). The preceding example is represented (after Pass Three) as

```
        ADD   B   A   C,
```

assuming that all the conditions for reduction are met.

The simplicity of STRIPE Operators is one of the more desirable features of STRIPE. The mapping from ULTRA/32 operators to STRIPE is quite straightforward. Even though the prototype of ZEBRA described in this report only encompasses 14 instructions out of the UYK-7 repertoire, the set and its mapping to STRIPE is exemplary of the repertoire in general and its mapping to STRIPE. The 14 were chosen for their frequency of use and their ability to demonstrate some of the domains of the utility of ZEBRA. STRIPE's flexibility illustrates a key point in the design of an intermediate representation for decompilation. Whenever the semantics of a source instruction cannot be successfully represented at a higher semantic level, the designer of a decompiler should be allowed to represent the semantics of that instruction at a lower level. A fundamental mistake of many previous decompiler designers has been to attempt to decompile every instruction of the source program to a high-level language. ZEBRA avoids that problem through the flexibility of STRIPE.

The locations of data and their initial values are preserved in STRIPE. This feature was included for several reasons. First, preserving this information adds to an understanding of the source program. The assumption made here is that the data were placed at their locations for valid reasons. Additionally, nothing is to be gained by listing all the data in one section of the program since a low-level language is the translation target, and program readability is an aim.

There are two fundamental goals that STRIPE was successfully designed to meet. First, it provides for the removal of machine dependency from the translated program. Independence from the mnemonics of ULTRA/32 grants a good measure of this, and elimination of the concept of LOAD and STORE is also effective in avoiding the constraints of the UYK-7 architecture. The other goal is to remove the operand implications from the program. In other words, all operands are explicitly referenced in STRIPE. This goal is worthwhile since it adds to the measure of liberation from one machine's architecture. For instance, the ULTRA/32 instruction

```
LA  1, DATA, 3
```

loads the contents of "DATA" into Accumulator Register 1. In Pass Two of ZEBRA, Accumulator Register 1 is given an ordinary variable name "$REGA1$$", an entry in the symbol table, a location at the end of the program, and an initial value of 0. Hence, in STRIPE, the instruction is represented as

```
ASN       DATA     <NULL>     $REGA1$$.
```

Since condition codes differ widely between computers, the ULTRA/32 COMPARE instruction is specially treated by Pass Two. The COMPARE instruction translates to no equivalent instruction in STRIPE, but its semantics are captured by the conditional JUMP instructions that follow in the same block. Specifically, the two operands being compared by the COMPARE instruction are explicitly named in each subsequent conditional JUMP instruction of the current block. For instance, the sequence

```
C   0, XYZ, 3              .  This is a comment.
JNZ    PLACE               .  This is, too.
```

translates to

```
<NONE>                        This is a comment.
JNZ $REGA0$$ XYZ PLACE        This is, too.
```

Note that comments are preserved. Any conditional JUMP instruction that does not have a COMPARE instruction preceding it in the same block is detected as an error condition.

Self-modifying code is detected by Pass Two and flagged as unacceptable. Self-modifying code is an area of a program acting as both data and instruction. It is detected by examining the destinations of instructions that store data into memory. If the symbols representing those destinations are catagorized as SIMPTRAN in the symbol table, then the destination is an instruction location. Elimination of self-modifying code is justified by the machine dependent semantics of such code.

PASS THREE - PROGRAM SIMPLIFICATION

Pass Three of ZEBRA reduces the STRIPE representation, where appropriate, by eliminating extraneous LOAD and STORE instructions. This step contributes to the production of machine independent programs by freeing them from the accumulator machine protocol. The program reduction is carried out on a block by block basis, but each prospective case of reduction is contingent upon information of a global scope. As before, program comments and symbol names are preserved through the simplification.

As an example of the simplification performed by Pass Three, consider the following block in STRIPE:

```
(1)   ASN    XYZ       <NULL>    $REGA1$$
(2)   ADD    ABC       $REGA1$$  $REGA1$$
(3)   ASN    $REGA1$$  <NULL>    DEF
(4)   JGT    $REGA1$$  LMN       GREAT
```

Pass Three first detects the use of an ASN instruction in line (1) whose destination is not redefined before it is used in subsequent line (2). Assuming that the global criteria for reduction (to be described later) are satisfactorily met, the source operand from the ASN instruction in line (1) is substituted for the occurrence of its destination as an operand in line (2), and line (1) is eliminated. This step yields

```
(2)   ADD    ABC       XYZ       $REGA1$$
(3)   ASN    $REGA1$$  <NULL>    DEF
(4)   JGT    $REGA1$$  LMN       GREAT
```

This type of substitution is called "forward" substitution because the line that was eliminated is behind the line in which the operand substitution took place.

Next, the destination of line (2) is discovered to be the source operand of a subsequent ASN instruction in line (3). Assuming, once again, that the reduction criteria are met, the destination operand of the ASN instruction in line (3) is substituted for the destination in line (2). In all subsequent lines before the destination operand of line (2) is redefined, occurrences of this symbol as source operands are substituted by the destination operand of line (3). Finally, line (3) is eliminated. This step yields the block

```
(2)   ADD    ABC       XYZ       DEF
(3)   JGT    DEF       LMN       GREAT
```

This step is called "backward" substitution because the line that was eliminated is in front of the line in which the destination substitution took place.

BUSY ANALYSIS

In accomplishing this reduction phase of ZEBRA, an important concept from compilation technology is utilized to help determine the appropriateness of reduction. This concept is called the "busy status" of variables. In ZEBRA, a variable is busy at some location if it is fetched before it is redefined past that location. For instance, in the following sequence of instructions, variable ABC is busy at line (1) because it is fetched at line (3) before it is redefined at line (3):

```
(1)   ASN    DEF    <NULL>   ABC
(2)   ADD    XYZ     DEF     DEF
(3)   SUB    ABC     DEF     ABC
```

In compiler optimization, a variable used within a block is often tested for its status relative to the "busy upon exit" attribute. The status is positive if that variable is busy in any of the immediate successors of the containing block or in any of those blocks' immediate successors. In compilers, that test is generally used to determine the appropriateness of reorganizing the order of basic blocks. In ZEBRA, that test is implemented by means of a recursive boolean function, and is used for program reduction. The function BUSY accepts as a parameter a list of immediate successor blocks in which to search for busy occurrences of the subject variable. Its execution begins with analysis of the first block numbered in the successor list. It scans this block, line by line. If the subject variable appears in any instruction line as a source operand before an instance of the variable as a destination operand, then the function returns a TRUE value. If, on the other hand, the variable appears as the destination operand of an instruction before it appears as a source operand, then the variable is redefined along its present flow path and, hence, cannot be busy along that path. If the variable is thus redefined, then the next block is taken from the immediate successor list passed to the function, and the scanning analysis begins on it. However, if there is no instance of the subject variable in this block, the function BUSY is called recursively with this block's immediate successor line as its parameter. Therefore, a depth-first search for incidences of a particular variable through block immediate successor lists is implemented.

The search is concluded and a value returned when a busy occurrence is found or when all the blocks in the immediate successor list and all their immediate successors have been scanned (if need be). The block numbers of blocks that have been scanned are kept in a list so that infinite recursion is impossible. Note that using this list also enhances the efficiency of the function. Once it is determined that scanning a certain instruction path leads to no busy occurrence of the subject variable, there is no reason to scan it again.

In addition to the function BUSY, ZEBRA incorporates a procedure BLKBSY, which generates a list of all busy occurrences of a subject variable within the current block beginning with a point specified to the procedure. These are called "block busy occurrences." The scan stops at the end of the block or when the subject variable is redefined.

BLOCK REDUCTION

Pass Three first sends, unchanged to the output file, all data lines and comment lines until it reaches the beginning of a block. It then inputs the entire block and begins scanning each instruction. If the operator is an ASN operator, then forward substitution is attempted. If it is any other non-JUMP operator that places a result into its destination operand, then backward substitution is attempted. In general, forward substitution is used for eliminating extraneous LOAD instructions, whereas backward substitution is used for extraneous STORE instructions. Finally, the block is output, and the algorithm is repeated until the entire program is finished.

Forward Substitution

This section describes the criteria for reducing the program by eliminating the current instruction, an ASN instructon, and substituting forward of this point. Let I represent the line number of this ASN instruction.

First, the reduction is not allowed if the destination of line I is an indexed variable (array variable). The symbol table contains this information. The reason for this exclusion is that it is impossible to tell, in general, whether an indexed variable is busy since the index value is determined at run-time. Next, a list of block busy locations past I of the destination operand of line I is generated by invoking procedure BLKBSY. As long as this list is not empty, the last block busy occurrence and each instruction within the block beyond the last occurrence are scanned to determine whether the destination operand of line I is redefined. This would indicate that the destination operand of line I is not busy past its last block busy occurrence. Let K represent the line number of the last block busy location of the destination of I. If this variable is not redefined within the block, then it is tested for the "busy upon exit" condition.

Assuming that the destination operand of line I has met all the criteria up to this point and that it is not busy upon exit from the current block, the source operand of line I is tested for type SIMPVAR. If it is tested, then the area of the block between I and K is scanned to ensure that the source operand of line I is not redefined. If the source operand of line I is not redefined, then line I meets all the criteria for elimination. The source operand of line I is substituted for each of the block busy occurrences of the destination of line I as recorded in the list, and line I is eliminated from the program.

There is still some chance for reduction even if the source operand of line I is of type ARRVAR. If there is no question concerning whether the value of the index could be changed between I and K, then the criteria for reduction are met. Therefore, if no instruction between I and K redefines the Index Register, reduction, as described in the preceding paragraph can occur.

Backward Substitution

This section describes the criteria for eliminating an ASN instruction located forward of the current instruction (a non-ASN instruction) and substituting back in the current instruction and forward of the eliminated instruction. Let I represent the line number of this current non-ASN instruction.

In this scheme, reduction is not allowed if the destination of line I is an indexed variable. The symbol table contains this information. This exclusion exists because it is impossible to tell, in general, whether an indexed variable is busy since the index value is determined at run-time. A list of block busy locations past I of the destination operand of line I is generated by invoking procedure BLKBSY. As long as this list is not empty, the first entry in the list (the one nearest I) that indicates an ASN instruction is found. If one exists, then it is a candidate for elimination. Call the location of this instruction K.

If the destination variable of line I is redefined within the block past K or if it is not busy upon exit, then the type of the destination of K is examined. In order still to be considered for regular elimination, its type must be SIMPVAR. Next a list of block busy occurrences of the destination of line K past I is compiled. If any entry in this list is between I and K, then substitution cannot occur because the destination of line K is being used for different purposes in this area than at K. Otherwise, backward substitution can occur. Backward substitution consists of replacing each occurrence of the destination operand of line I in its block busy list with the destination of line K (the ASN instruction). Next, the actual destination operand of line I is replaced with the destination operand of line K and line K is eliminated.

There is still some chance for reduction even if the destination operand of line K is of type ARRVAR. Special case reduction is indicated if no instruction between I and the last block busy occurrence of the destination of I redefines the Index Register. Reduction is carried out as described in the preceding paragraph.

CONCLUSIONS

One method of achieving a certain measure of portability for assembly language software has been described. This report does not imply that decompilers should or (even practically) could be built that capture the semantics of every program written in the source assembly language. On the contrary, decompiling is, in general, an incomplete process. While it may be possible to transport an arbitrary program completely through decompilation, it is not economically practical to do so. Programmer tricks and special cases raise the complexity of total decompilation to exorbitant levels. Decompilation is meant to be an aid to the transportation process, not to be the entire process itself.

A factor that greatly affects the degree of translation achieved by a decompiler is the level of the decompilation target language. If it can represent low-level constructs such as shift and mask, then more of the software can be captured than through the use of a language in which such constructs have no meaning. For instance, a circular shift has no meaning in a high-level language such as PASCAL. Flexibility of the decompilation target language should be its overriding aim.

The target level of decompilation in ZEBRA seems to be both necessary and sufficient to achieve its goals relative to portability. After analyzing several programs typical of the UYK-7 applications, it was determined that the Load Accumulator and Store Accumulator instructions are generally used much more often than any other instruction in the UYK-7 repertoire. In fact, it was observed that the Load Accumulator instruction was used more than twice as often as the fourth most often used instruction. (The Unconditional JUMP instruction was ranked third.) Therefore, eliminating uses of these instructions when they are extraneous to the semantics of the program contributes substantially to the space efficiency of the decompilation target program. In fact, Housel,[11] in implementing a similar scheme for eliminating extraneous LOAD and STORE instructions observed, 'For the samples tested, the text compression process reduces the "volume" (no. of instructions) of the program up to 40 percent.'

As an intermediate representation, STRIPE is successful because of its flexibility and simplicity. Its further use is encouraged by the structure of the assembly languages of currently popular machines. Computer architects are striving to raise the semantic levels of their computers' native languages. For instance, Digital Equipment Corporation's VAX 11/780 computer has an ADD instruction which duplicates the STRIPE ADD instruction in structure and interpretation. This trend toward higher level native languages should continue.

Use of a decompilation system such as ZEBRA has certain fringe benefits aside from the obvious ones. In addition to the benefits gained by translating a program to another language such as the ability to perceive it from a new point of view, ZEBRA provides a control flow graph of the program. If this ability were coupled with some additional software, then an automatic documentation and an automatic flowcharting capability would be added to the system. This capability would be quite useful in maintaining old software.

A few words should be said about the efficiency of decompilation as considered in the design phase. Since a decompiler is intended to be an aid to portability, it is not likely to be executed more than one or two times for a given source program (as opposed to the several times a compiler being used to develop a program would be executed). Therefore, where tradeoffs arise, the decompiler should prefer space economy over execution time economy because it is always taking up space somewhere, but is rarely being executed. However, the product of decompilation must be easily understandable because program development may continue after translation. Hence, clarity issues should take precedence over both time and space issues.

In designing this prototype of ZEBRA, several features were omitted with the understanding that future revisions would possibly include them. At first, of course, more instructions must be added to those handled by ZEBRA along with the data types supported by ULTRA/32. Addition of these instructions and data types is a strightforward programming task because of the modularity of ZEBRA and the flexibility of STRIPE. ZEBRA should also be modified to support more of the ULTRA/32 assembler directives.

The base registers of the UYK-7 can be supported by ZEBRA by appending the register number, which appears in an instruction S subfield, to the name of the symbol appearing in the Y subfield of the instruction before entering it or accessing the symbol in the symbol table. The section dealing with Pass Two discusses this solution.

The K subfield designator is best accounted for directly in the structure of STRIPE. Since the K subfield indicates data type information about the operand of an instruction, its information should be represented at the highest semantic level. In other words, data type information is, by nature, at a rather high level, and the highest level of representation in ZEBRA is STRIPE. Hence, data type descriptors should be added to the operand information kept in each record of STRIPE in order to fully support the ULTRA/32K subfield.

Indexed jumps are currently flagged by Pass One on the grounds that the capability they give the language for transfer of control based on dynamic parameters compromises the integrity of the control flow analysis. However, if the range of values that can be attained by the Index Register is limited to some small set, then that information can be contributed to the system and the flow graph modified accordingly. The Index Register range information can perhaps be generated from a global analysis of operand instances of the Index Register. An alternative way to get that information is to ask the user. However, that implies a very smooth interface between the translator and the user.

Indirection in JUMP instructions is precluded for the same reasons as indexed JUMPs. Without significantly more analysis, it is impossible to limit the number of possible JUMP destinations. Indirection with STORE instructions renders the detection of self-modifying code impossible, unless the range of attainable values of the indirect word is limited. These limits can possibly be discovered through a global analysis of tne indirect word or by asking the user.

Finally, a code generator from STRIPE to the target assembly language should be built in order to complete the translation. There is no reason for this code generator to be any different in principle from code generators commonly built for compilers. STRIPE's similarity to many common intermediate languages of compilers simplifies the task of implementing the code generator.

REFERENCES

1. T. B. Steel, Jr., "A First Version of UNCOL, Procedings AFIPS WJCC, Vol. 19, 1961, pp. 371-377.

2. B. C. Housel, "A Study of Decompiling Machine Languages," CSC TR100, Computer Science Department, Purdue University, Lafayette, IN, August 1973.

3. B. C. Housel and N. H. Halstead, "A Methodology for Machine Language Decompilation," Computer Science RJ1316 (020557), T.J. Watson Research Center, Yorktown Heights, NY, December 1973.

4. J. C. Warren, Jr., "Software Portability, A Summary of Related Concepts and Survey of Problems and Approaches," Technical Note No. 48, Digital Systems Laboratory, Stanford University, Stanford, CA, September 1974.

5. M. Richards, "The Portability of the BCPL Compiler," Software - Practice and Experience, Vol. 1, 1971, pp. 135-146.

6. J. M. Newcomer, "Machine Independent Generation of Optimal Local Code," Doctoral Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, May 1975.

7. P. L. Miller, "Automatic Creation of a Code Generator from a Machine Description," MAC TR 85, Massachusetts Institute of Technology, Cambridge, MA, May 1971.

8. P. J. Orgass and W. M. Waite, "A Base for a Mobile Programming System," Communications of the ACM, Vol. 12, No. 9, September 1969, pp. 507-510.

9. W. M. Waite, "The Mobile Programming System : STAGE2," Communications of the ACM, Vol. 13, July 1970, pp. 415-442.

10. P. Barbe, "The Piler System of Program Translation," PLR-020, Probe Consultants, Inc., Phoenix, AZ, September 1974.

11. C. R. Hollander, "Decompilation of Object Programs," Technical Report 54, Digital Systems Laboratory, Stanford University, Stanford, CA, January 1973.

12. C. R. Hollander, "A Syntax-Oriented Formal System for Defining Processes," Procedings of the ACM National Conference, Association for Computing Machinery, New York, 1973, pp. 295-298.

REFERENCES (Cont'd)

13. A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, MA, 1977.

14. D. Gries, Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, 1971.

INITIAL DISTRIBUTION

Addressee	No. of Copies
NAVSEA (SEA-92R, PMS-409, PMS-408 (CDR Goodman))	3
NAVAIRDEVCEN (Code 2052)	1
NOSC (Library, Code 6565)	1
DTNSRDC	1
NAVSURFWPNCEN	1
NPS, Monterey	1
NAVMAT (MAT-08Y, O. McOmber)	1
DDC	2